CSCI 365 Problem Set 5: Higher-Order Functions and Recursion Patterns

due Friday, 23 February 2024

Specification

To receive credit for this problem set:

- You must complete at least 8 out of 11 exercises.
- Any code you write must adhere to the Haskell style guide linked from the course web page.

Haskell code should be submitted in one or more `.hs` files. Written exercises may be submitted either as a PDF, or as comments in one of the `.hs` files.

Idiomatic Haskell

Exercise 1 Here is a list of standard library functions that every Haskell programmer ought to be familiar with:

This list is not exhaustive by any means but it's a good starting point!

```
fst snd max min even odd
flip (.) show read
head last tail init (++) null concat
map foldr concatMap
product sum and or all any length maximum minimum
filter find
iterate repeat replicate cycle take drop splitAt takeWhile dropWhile
reverse sort
zip zipWith unzip
maybe either
```

Pick three functions from the above list that you are unfamiliar with. At least one must be a higher-order function. For each function, do the following:

1. Read the documentation for the function at `https://hackage.haskell.org/package/base/docs/Prelude.html`.
2. Write down the function's type and explain in your own words what the function does.[1]
3. Give an example expression which uses the function.

[1] For maximum learning, try this: read the function's documentation, then put it aside and do something else for five minutes. Then try to write down as much about the function as you can remember, without peeking (its type, a description of what it does, *etc.*). Then check to see how accurate you were and whether you missed anything.

Higher-order functions

Exercise 2 Without looking at its implementation in the standard library, implement the function

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

which returns a prefix of the given list, stopping just before the first element which does not satisfy the predicate. For example,

- `takeWhile (<5) [1..10] = [1, 2, 3, 4]`
- `takeWhile isLower "camelCase" = "camel"`
- `takeWhile (>= 0) [1,2,3] = [1,2,3]`
- `takeWhile (< 0) [1,2,3] = []`

Exercise 3 Implement a function

```
xor :: [Bool] -> Bool
```

which returns `True` if and only if there are an odd number of `True` values contained in the input list. It does not matter how many `False` values the input list contains. For example,

```
xor [False, True, False] == True
xor [False, True, False, False, True] == False
```

Your solution **must be implemented using `foldr`**, that is, it should be of the form

```
xor = foldr ...
```

Exercise 4 Complete CFP Exercise 10.9 (`iter`). Bonus points for using `id` and `(.)` in your solution.

Exercise 5 Complete CFP Exercise 11.3 (`composeList`).

© 2024 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

Exercise 6 Recall the definition of a *binary tree* data structure. The *height* of a binary tree is the length of a path from the root to the deepest node. For example, the height of a tree with a single node is 0; the height of a tree with three nodes, whose root has two children, is 1; and so on. A binary tree is *balanced* if the heights of its left and right subtrees differ by no more than 1, and its left and right subtrees are also balanced.

http://en.wikipedia.org/wiki/Binary_tree

You should use the following data structure to represent binary trees. Note that each node stores an extra `Integer` representing the size (total number of nodes) of the binary tree at that node.

```
data Tree a = Empty
            | Node Integer (Tree a) a (Tree a)
  deriving (Show, Eq)
```

For this exercise, write a function, **implemented using `foldr`**,

```
mkBalancedTree :: [a] -> Tree a
mkBalancedTree = foldr ...
```

which generates a balanced binary tree from a list of values.

For example, one sample output might be the following, also visualized at right in Figure 1:

```
foldTree "ABCDEFGHIJ" ==
  Node 10
    (Node 4
      (Node 1 Leaf 'F' Leaf)
      'I'
      (Node 2 (Node 1 Leaf 'B' Leaf) 'C' Leaf))
    'J'
    (Node 5
      (Node 2 (Node 1 Leaf 'A' Leaf) 'G' Leaf)
      'H'
      (Node 2 (Node 1 Leaf 'D' Leaf) 'E' Leaf))
```

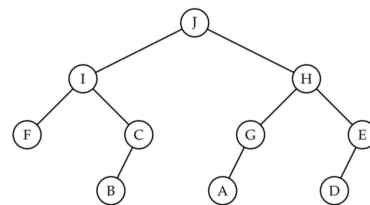

Figure 1: A balanced tree

Your solution might not place the nodes in the same exact order, but it should result in a balanced tree, with each subtree having a correct computed size.

© 2024 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

Wholemeal programming

Exercise 7 Complete CFP Exercise 10.2 (`length` via `map` and `sum`).

Exercise 8 A *local maximum* of a list is an element of the list which is strictly greater than both the elements immediately before and after it. For example, in the list `[2,3,4,1,5]`, the only local maximum is `4`, since it is greater than the elements immediately before and after it (`3` and `1`). `5` is not a local maximum since there is no element that comes after it.

Write a function

```
localMaxima :: [Integer] -> [Integer]
```

which finds all the local maxima in the input list and returns them in order. For example:

```
localMaxima [2,9,5,6,1] == [9,6]
localMaxima [2,3,4,1,5] == [4]
localMaxima [1,2,3,4,5] == []
```

Exercise 9 Write a function

```
histogram :: [Integer] -> String
```

which takes as input a list of `Integers` between 0 and 9 (inclusive), and outputs a vertical histogram showing how many of each number were in the input list. You may assume that the input list does not contain any numbers less than zero or greater than 9 (that is, it does not matter what your function does if the input does contain such numbers). Your output must exactly match the output shown in the examples below.

© 2024 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

```
histogram [1,1,1,5] ==

 *
 *
 *   *
==========
0123456789

histogram [1,4,5,4,6,6,3,4,2,4,9] ==

    *
    *
    * *
 ******  *
==========
0123456789
```

Important note: If you type something like `histogram [3,5]` at the `ghci` prompt, you should see something like this:

```
"   * *    \n==========\n0123456789\n"
```

This is a textual *representation* of the `String` output, including `\n` escape sequences to indicate newline characters. To actually visualize the histogram as in the examples above, use `putStr`, for example, `putStr (histogram [3,5])`.

Further Exploration

Exercise 10 Read sections 1–5 of John Backus's 1977 Turing Award lecture, *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, available from `http://worrydream.com/refs/Backus-CanProgrammingBeLiberated.pdf`.

Backus delivered this lecture 45 years ago. In what ways do his remarks still apply today, and in what ways are they outdated? Give two specific examples of each.

Also, translate the Innerproduct function, defined in section 5.2, into Haskell. (You may use the standard `transpose` function, defined in the `Data.List` module.)

The rest of this lecture has lots of cool stuff in it too; you're welcome to read more if you find it interesting, although beware that it starts getting extremely hairy about halfway through.

Exercise 11 Read at least sections 1–4 of Ralf Hinze's Functional Pearl from ICFP 2009, *La Tour D'Hanoï*,[2] which can be found at `https://www.cs.ox.ac.uk/ralf.hinze/publications/ICFP09.pdf`, and try implementing some of the functions Hinze describes.

[2] Ralf Hinze. 2009. Functional pearl: la tour d'Hanoï. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09). Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/1596550.1596555

© 2024 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.